FTML practical session 4

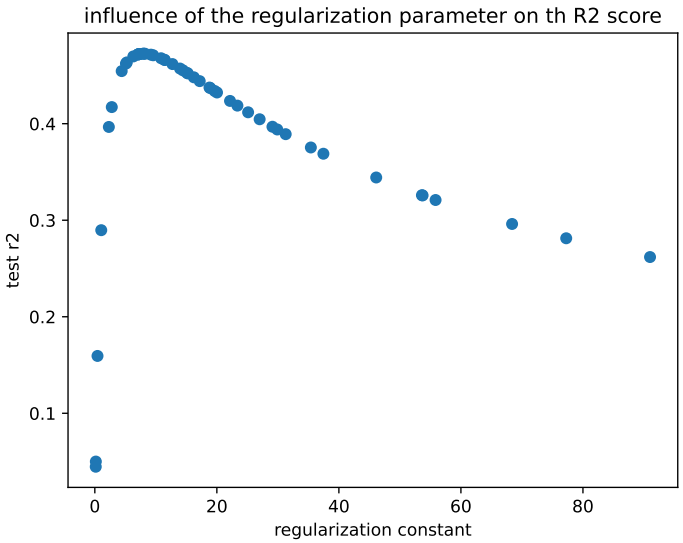

TABLE DES MATIÈRES

1 HYPERPARAMETER TUNING METHODS

In this exercise we summarize the most common hyperparameter tuning methods and use **optuna** as a bayesian optimization framework.

1.1 Grid search and random search

Grid search is the most basic approach to hyperparameter tuning : choose a list of values for each and iterate through a grid containing all possible combinations.
https://scikit-learn.org/stable/modules/grid_search.html#

See also :

— random search

— successive halving.

1.1.1 *Issues*

Grid search might be suboptimal in the sense that all hyperparameter combinations will be tried, although many of them might be irrelevant. Also, the size of the grid is exponential in the number of parameters.

1.1.2 *Question*

Why can we usually not run a gradient-based optimization algorithm to look for better HP values ?

1.2 Bayesian optimization

More modern approaches use smarter optimization methods to tune the hyperparameters. For instance, **Bayesian optimization**.

https://en.wikipedia.org/wiki/Bayesian_optimization

1.2.1 *Frameworks*

Several frameworks exist to perform Bayesian optimization over HPs values, and automate the search.

skopt https://scikit-optimize.github.io/stable/

optuna https://optuna.org/

With these methods, we can specify ranges for scalar HP values, instead of grids of values.

1.2.2 *Optuna and optuna dashboard*

Optuna creates "studies", in which HP sets are tested in a sequence. A set of HPs is tested in a "trial".

https://optuna.readthedocs.io/en/stable/reference/study.html

Some interesting optuna aspects to consider :

— some trials can be "pruned" : they are considered as not interesting by optuna, and are discarded before the optimization is finished.

— it is possible to start a study, stop the program, and resume the study later, with the previous trials still in the study database.

— you can analyze the results of a study by processing the study object with other libraries like pandas, seaborn, matplotlib, etc.

With optuna, you can display and analyze the influence of the HP with a dashboard that opens in your browser.

`https://github.com/optuna/optuna-dashboard`

Perform HP optimization with optuna over a problem of your choice. Visualize the results in the optuna dashboard

Suggestion : if you want, you can use **main_optuna_template.py** and the data contained in **data/**, in **exercise_3_optuna/** folder, but you are encouraged to tru also in a different problem, for instance with scikit's toy datasets or any dataset of your choice.

2 CURSE OF DIMENSIONALITY AND ADAPTIVITY TO A LOW DIMENSIONAL SUPPORT

2.1 Nearest neighbors algorithms

A nearest neighbors algorithm is a method of prediction, based on a dataset, but without any optimization. It consists in averaging the predictions of the nearest neighbors of a sample x in a given dataset. See two examples below, where both input and output spaces are $\mathbb{R}$. The two example differ by the number of samples in the dataset.

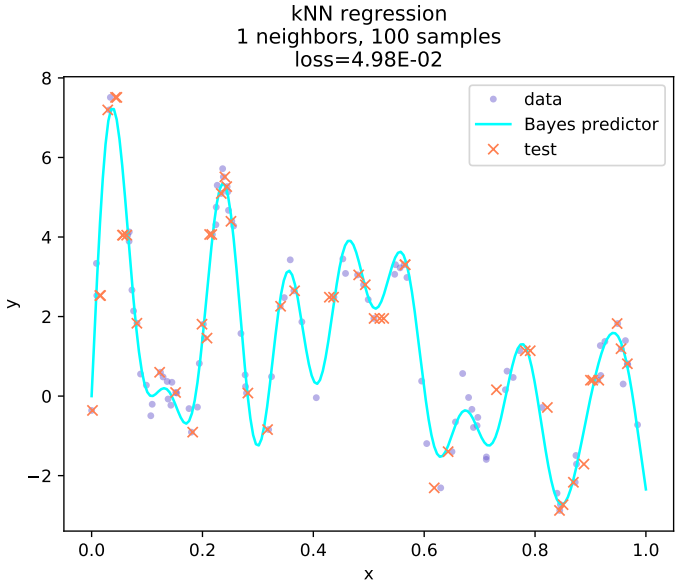

FIGURE 1 – knn

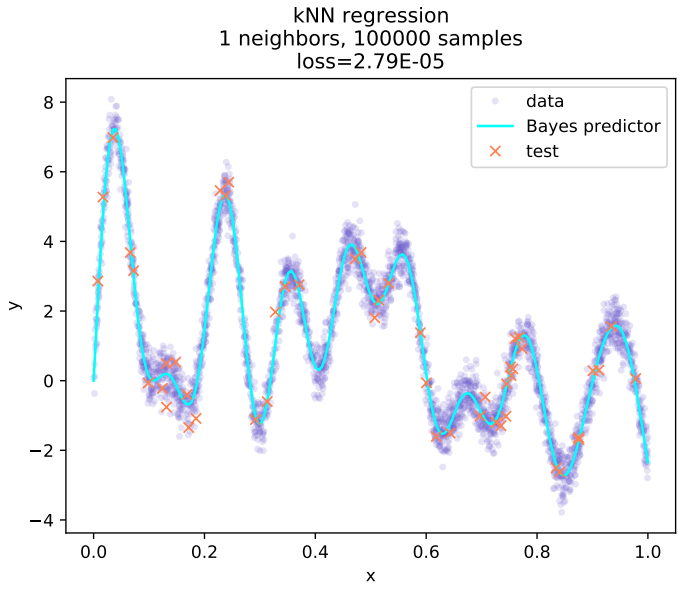

FIGURE 2 – knn

A natural question would be : why do we not use this method all the time ? The answer is that as soon as the input space dimension is not very small (for instance not < 10), the number of samples that would be required to obtain a satisfactory error is simply too large.

2.2 Curse of dimensionality : general case

In the general case, it is possible to show that if the target function f^* is Lipshitz-continuous, and if the n samples in the dataset are on a lattice that divides $[0,1]^d$ in cubes of equal size, then the error made by a k-NN esimator can be upper bounded by a bound of the form $\mathcal{O}(n^{-1/d})$.

Note that this is a rough bound. It is possible to have more subtle results, and the constant in the $\mathcal{O}$ depends on d and on the Lipshitz constant of the target function However, it is not possible to have a bound that is "way better" (i.e. "way smaller") than this bound.

This means that if d is large, then $1/d$ is small, and the decrease of the error is very slow. To be more precise, in order to reach an error ϵ, it is possible to show that the number of sampled needed n_ϵ verifies

$$n_\epsilon \geqslant \frac{\epsilon^{-d} d^{d/2}}{\alpha^{d/2}} \tag{1}$$

where $\alpha > 0$ is a constant. n_ϵ hence grows exponentially with $1/\epsilon$, and more than exponentially with d which is a example of curse of dimensionality.

We note that our target function is indeed Lipshitz continuous and thus respect the hypothesis to obtain equation 1. If a function is not Lipshitz continuous, the situation is even worse !

2.2.1 *Simulation : uniformly distributed data in the input space*

We will study the influence of the input space dimension on the possibility to approximate a function with a nearest neighbors approach in a specific setting :

— input space : $\mathcal{X} = [0,1]^d$. Hence, here the dimensionality of the problem is d. The data will be uniformly sampled in the input space.

— output space : $\mathcal{Y} = \mathbb{R}$

— toy function to approximate : $f : x \in \mathbb{R}^d \mapsto \|x\|$ (euclidean norm).

Run a simulation that computes the test error of a kNN estimator, for different values of n and d, in order to observe the curse of dimensionality. The dataset should be uniformly sampled in the input space. You can experiment with the number of neighbors used of k, for instance 2.

You can use the template file **knn_uniform.py**. Note that as we know the function that we try to approximate, it is possible to perfeclty evaluate the error made by the k-NN algorithm for each sample. For instance, you can observe images like the one in figure 3. In the simulation that generated the figure, the samples (both the dataset and the test samples) were drawn randomly in $\mathcal{X}$, with a uniform distribution (hence the dataset is not on a regular lattice, although you can do use a regular lattice in your simulation).

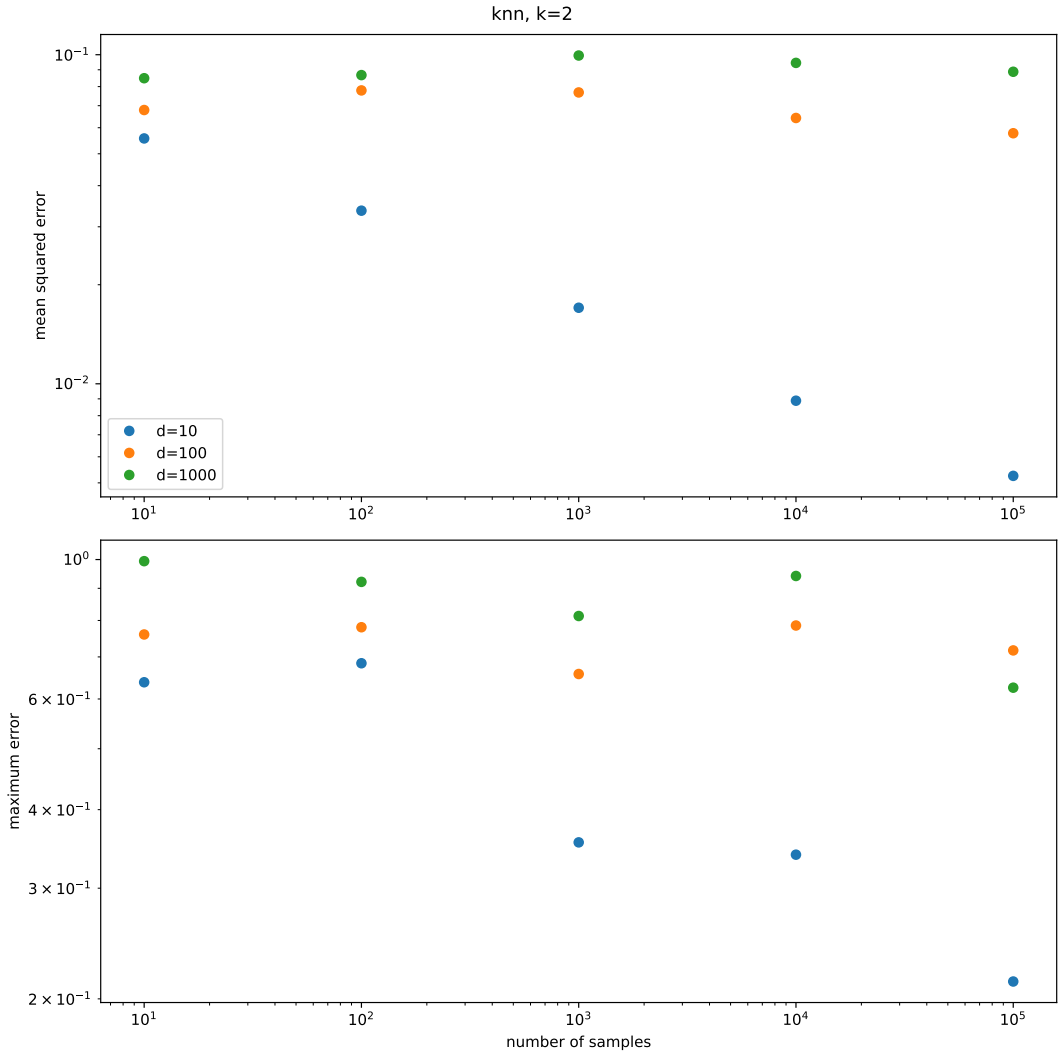

FIGURE 3 – Curse of dimensionality. We observe that the logarithm of the error is linear in the logarithm of the number of samples, with a slope that depends on the dimension d. When d is large, the problem is that the curve is very flat, which means that the error decreases very slowly.

2.3 Adaptivity

However, the situation is not always that catastrophic in high dimensions and we will illustrate this with an example. We keep exactly the same setting, but now the

input dataset X is given and fixed, and we use $d = 30$.

Perform a k-NN estimation using the data stored in **data_knn/** and use a varying number of samples n that you use from the dataset. Plot the error as a function of n.

You can use the template file **knn_adaptivity.py**. You should observe something like figure 4. We note that the learning rate is way better than $\mathcal{O}(n^{-1/d})$.

— What could be an explanation of this phenomenon ?

— How could we test this hypothesis ? (we have seen a way during the class)

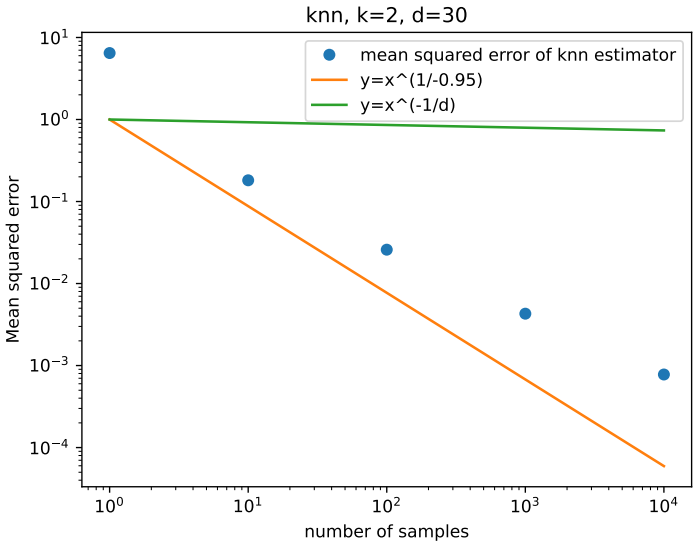

FIGURE 4 – Nearest neighbors estimation on a different dataset.